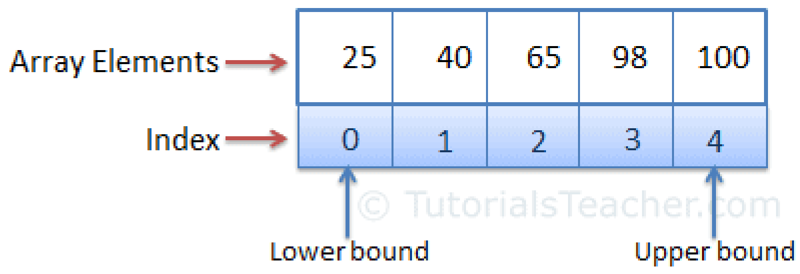
Arrays

- A variable is used to store a literal value, whereas an array is used to store multiple literal values.
- An array is the data structure that stores a fixed number of literal values (elements) of the same data type. Array elements are stored contiguously in the memory.
- In C#, an array can be of three types:
- single-dimensional, multidimensional, and jagged array.
- The following figure illustrates an array representation.



Array Representation

Array Declaration and Initialization

- An array can be declared using by specifying the type of its elements with square brackets.

Example: **Array Declaration**

- int[] evenNums; // integer array
- string[] cities; // string array

Example: **Array Declaration & Initialization**
- int[] evenNums = new int[10]{ 2, 4, 6, 8, 10,7,90,00 };
- string[] cities = new string[3]{ "Mumbai", "London", "New York" };

Arrays type variables can be declared using var without square brackets.

Example: **Array Declaration using var**
var evenNums = new int[]{ 2, 4, 6, 8, 10};
var cities = new string[]{ "Mumbai", "London", "New York" };

Example: **Short Syntax of Array Declaration**
int[] evenNums = { 2, 4, 6, 8, 10};
string[] cities = { "Mumbai", "London", "New York" }

Example: **Invalid Array Creation**
```
//must specify the size
int[] evenNums = new int[];
```

```
//number of elements must be equal to the specified size
int[] evenNums = new int[5] { 2, 4 };

//cannot use var with array initializer
var evenNums = { 2, 4, 6, 8, 10};
```

Late Initialization

- It is not necessary to declare and initialize an array in a single statement.
- You can first declare an array then initialize it later using the new operator.

Example: **Late Initialization**

```
int[] evenNums;//declaration part

evenNums = new int[5];
// or
evenNums = new int[]{ 2, 4, 6, 8, 10 };
```

Accessing Array Elements

- Array elements can be accessed using an index.
- An index is a number associated with each array element, starting with index 0 and ending with array size - 1.

Example: **Access Array Elements using Indexes**

```
int[] evenNums = new int[7];
evenNums[0] = 2;
evenNums[1] = 4;
evenNums[2] = 4;
evenNums[3] = 4;
evenNums[4] = 4;
//evenNums[6] = 12;  //Throws run-time exception IndexOutOfRange
Console.WriteLine(evenNums[0]);  //prints 2
Console.WriteLine(evenNums[1]);  //prints 4
```

Accessing Array using for Loop

- Use the `for` loop to access array elements. Use the `length` property of an array in conditional expression of the for loop.

Example: **Accessing Array Elements using for Loop**

```
int[] evenNums = { 2, 4, 6, 8, 10 };
for(int i = 0; i < evenNums.Length; i++)
Console.WriteLine(evenNums[i]);
for(int i = 0; i < evenNums.Length; i++)
   evenNums[i] = evenNums[i] + 10;  // update the value of each element by 10
```

Accessing Array using foreach Loop

- Use foreach loop to read values of an array elements without using index.

Example: **Accessing Array using foreach Loop**

```
int[] evenNums = { 2, 4, 6, 8, 10};
string[] cities = { "Mumbai", "London", "New York" };
foreach(var item in evenNums)
Console.WriteLine(item);
foreach(var city in cities)
Console.WriteLine(city);
```

LINQ Methods

- All the arrays in C# are derived from an abstract base class System.Array.
- The Array class implements the IEnumerable interface, so you can LINQ extension methods such as Max(), Min(), Sum(), reverse(), etc. See the list of all extension methods here.

Example**: LINQ Methods**

```
int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

nums.Max(); // returns 16
nums.Min(); // returns 6
nums.Sum(); // returns 55
nums.Average(); // returns 55
```

- The System.Array class also includes methods for creating, manipulating, searching, and sorting arrays. See list of all Array methods here.

Example: **Array Methods**

```
int[] nums = new int[5]{ 10, 15, 16, 8, 6 };

Array.Sort(nums); // sorts array
Array.Reverse(nums); // sorts array in descending order
Array.ForEach(nums, n => Console.WriteLine(n)); // iterates array
Array.BinarySearch(nums, 5);// binary search

Int x , int y;

X=50;
Y=67;
Addition(x,y);
Int addition(int a , int b)
{
 Int c =a+b;
}
```

Passing Array as Argument

- An array can be passed as an argument to a method parameter.
- Arrays are reference types, so the method can change the value of the array elements.

Example: **Passing Array as Argument**

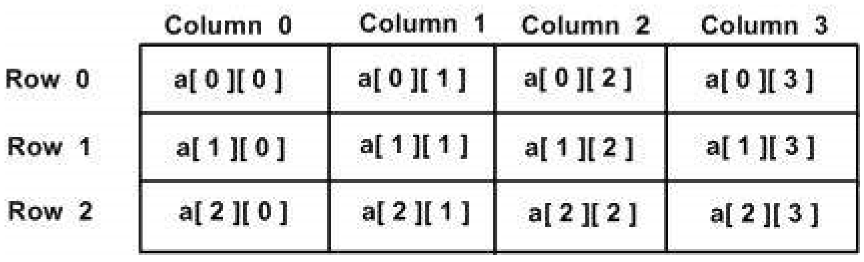
```
public static void Main(){
   int[] nums = { 1, 2, 3, 4, 5 };
   UpdateArray(nums);
   foreach(var item in nums)
      Console.WriteLine(item);
}

public static void UpdateArray(int[] arr)
{
   for(int i = 0; i < arr.Length; i++)
      arr[i] = arr[i] + 10;
}
```

- C# allows multidimensional arrays. Multi-dimensional arrays are also called rectangular array.

Two-Dimensional Arrays

- The simplest form of the multidimensional array is the 2-dimensional array.
- A 2-dimensional array is a list of one-dimensional arrays.
- A 2-dimensional array can be thought of as a table, which has x number of rows and y number of columns.
- Following is a 2-dimensional array, which contains 3 rows and 4 columns –



	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

- Every element in the array a is identified by an element name of the form a[i , j],
- Here a is the name of the array, and i and j are the subscripts that uniquely identify each element in array a.

Initializing Two-Dimensional Arrays

- Multidimensional arrays may be initialized by specifying bracketed values for each row.
- The Following array is with 3 rows and each row has 4 columns.

```
int [,] a = new int [3,4] {
  {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
  {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
  {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

```
a[1,1][1,2][1,3][1,4]
a[2,1][2,2][2,3][2,4]
```

Accessing Two-Dimensional Array Elements

- An element in 2-dimensional array is accessed by using the subscripts.
- row index and column index of the array.

```
int val = a[2,3];
```

```
using System;
namespace ArrayApplication {
  class MyArray {
    static void Main(string[] args) {
      /* an array with 5 rows and 2 columns*/
      int[,][,] a = new int[5, 2] {{0,0}, {1,2}, {2,4}, {3,6}, {4,8} };
      int i, j;

      /* output each array element's value */
      for (i = 0; i < 5; i++) {

        for (j = 0; j < 2; j++) {
          Console.WriteLine("a[{0},{1}] = {2}", i, j, a[i,j]);
        }
      }
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result –

```
a[0,0]: 0
a[0,1]: 0
a[1,0]: 1
a[1,1]: 2
a[2,0]: 2
a[2,1]: 4
a[3,0]: 3
a[3,1]: 6
a[4,0]: 4
```

```
a[4,1]: 8
```

Jagged Array

- Jagged array is an array of arrays.
- You can declare a jagged array named *scores* of type **int** as −

Initialize a jagged array

- int[][] scores = new int[2][]{new int[]{92,93,94},new int[]{85,66,87,88}};
- Here, scores is an array of two arrays of integers - scores[0] is an array of 3 integers and scores[1] is an array of 4 integers.

Example

```
using System;
namespace ArrayApplication {
  class MyArray {
    static void Main(string[] args) {

      /* a jagged array of 5 array of integers*/
      int[][] a = new int[][]{new int[]{0,0},new int[]{1,2},
        new int[]{2,4},new int[]{ 3, 6 }, new int[]{ 4, 8 } };
      int i, j;

      /* output each array element's value */
      for (i = 0; i < 5; i++) {
        for (j = 0; j < 2; j++) {
          Console.WriteLine("a[{0}][{1}] = {2}", i, j, a[i][j]);
        }
      }
      Console.ReadKey();
    }
  }
}
```

When the above code is compiled and executed, it produces the following result −

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

- Pass an array as a function argument in C#.

```
using System;
namespace ArrayApplication {
  class MyArray {
    double getAverage(int[] arr, int size) {
      int i;
      double avg;
      int sum = 0;

      for (i = 0; i < size; ++i) {
        sum += arr[i];
      }
      avg = (double)sum / size;
      return avg;
    }
    static void Main(string[] args) {
      MyArray app = new MyArray();

      /* an int array with 5 elements */
      int [] balance = new int[]{1000, 2, 3, 17, 50};
      double avg;

      /* pass pointer to the array as an argument */
      avg = app.getAverage(balance, 5 ) ;

      /* output the returned value */
      Console.WriteLine( "Average value is: {0} ", avg );
      Console.ReadKey();
    }
  }
}
```

ref Modifier

- By default, a reference type passed into a method will have any changes made to its values reflected outside the method as well.
- If you assign the reference type to a new reference type inside the method, those changes will only be local to the method.
- Using the **ref modifier**, you have the option to assign a new reference type and have it reflected outside the method.

Example

```
class ReferenceTypeExample
{
  static void Enroll(ref Student student)
  {
    // With ref, all three lines below alter the student variable outside the method.
    student.Enrolled = true;
    student = new Student();
    student.Enrolled = false;
  }
  static void Main()
  {
    var student = new Student
    {
      Name = "Susan",
      Enrolled = false
    };
    Enroll(ref student);
    // student.Name is now null since a value was not passed when declaring new Student() in
the Enroll method
    // student.Enrolled is now false due to the ref modifier
  }
}
public class Student {
  public string Name {get;set;}
  public bool Enrolled {get;set;}
}
```

- Using the **ref modifier**, you can also change value types outside the method as well.

```
class ReferenceTypeExample
{
  static void IncrementExample(ref int num)
  {
    num = num + 1;
  }
  static void Main()
  {
    int num = 1;
    IncrementExample(ref num);
    // num is now 2
  }
}
```

out Modifier

- Using the **out modifier**, we initialize a variable inside the method.
- Like **ref**, anything that happens in the method alters the variable outside the method.
- With **ref**, you have the choice to *not* make changes to the parameter.
- When using **out**, you must initialize the parameter you pass inside the method.

```
class ReferenceTypeExample
{
  static void Enroll(out Student student)
  {
    //We need to initialize the variable in the method before we can do anything
    student = new Student();
    student.Enrolled = false;
  }
  static void Main()
  {
    Student student;
    Enroll(out student); // student will be equal to the value in Enroll. Name will be null and
Enrolled will be false.
  }
}

public class Student {
  public string Name {get;set;}
  public bool Enrolled {get;set;}
}
```

- The **out modifier** works with value types as well. A useful example is using the out modifier to change a string to an int.

```
int x;
Int32.TryParse("3", out x);
```